Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited

Introduction

This package provides implementations of the main-memory sort-merge join algorithms described and studied in our VLDB 2013 paper (presentation in VLDB 2014). In addition to the newly introduced sort-merge join algorithms **m-way** and **m-pass**, the code package also provides implementations for various algorithms such as single-threaded AVX sorting routine and multi-way merge as well as various micro-benchmarks. The essential sort-merge algorithms presented in the paper are the following:

- **m-pass**: Sort-merge join with multi-pass naive merging
- **m-way**: Sort-merge join with multi-way merging

Note

For the implementation of the hash join algorithms mentioned in the paper, please obtain the hash joins code package from our webpage.

The implementations are targeted for the first version of the AVX instruction set. The AVX initially did not provide instructions for integers. Therefore, our code treated 2x32-bit integers as 64-bit doubles and operated on them with the corresponding AVX instructions: <sign:1bit-exponent:11bits-fraction:52bits> = <key:32-bits-payload:32-bits>. While this provided correct results for our study, it might lead to errors in generic cases such as when NaN patterns are found in real data. For our case, this can only occur when the key contains all 1s for bits 20:30 and when either of key[0:19] and payload[0:31] has a non-zero bit. Our generators carefully eliminate this from the inputs. In the latest version of AVX (i.e., AVX2), this problem will be completely eliminated.

Warning

The code has been tested and experimentally evaluated on the **Intel E5-4640**, a four socket Sandy Bridge machine with 2.4 Ghz clock, 512 GiB memory and AVX support. The OS is Debian Linux 7.0, kernel version 3.4.4-U5 compiled with transparent huge page support. The OS is configured with 2 MiB VM pages for memory allocations transparently. For reproducing the results from the paper all of these parameters should ideally be matched. Moreover, the NUMA-topology of the underlying machine also plays and important role. If the machine has a different NUMA-topology than the E5-4640, then some changes in the code and/or in cpu-mappings.txt might be required. Moreover, memory usage was not the primary concern of our paper and therefore algorithms are usually implemented in an out-of-place manner which requires extensive memory usage. Lack of sufficient memory (3-4X more than input relations) might result in unexpected performance behavior. Please contact us if you need more information regarding how to best configure the code for a given machine.

Compilation

The package includes implementations of the algorithms and also the driver code to run and repeat the experimental studies described in the paper.

The code has been written using standard GNU tools and uses autotools for configuration. Thus, compilation should be as simple as:

```
$ ./configure
$ make
```

Note

For the baseline comparisons, C++ STL sort algorithm is used only if the code is compiled with a C++ compiler. In order to enable this, configure with **`./configure CC=g++`**.

Besides the usual ./configure options, compilation can be customized with the following options:

```
--enable-key8B  use 8B keys and values making tuples 16B [defa
ult=no]
--enable-perfcounters  enable performance counter monitoring w
ith Intel PCM  [default=no]
--enable-debug  enable debug messages on commandline  [default
=no]
--enable-timing  enable execution timing  [default=yes]
--enable-materialize  Materialize join results ?  [default=no]
--enable-persist  Persist input tables and join result table t
o files (R.tbl S.tbl Out.tbl) ?  [default=no]
```

Note

If the code is compiled with **–enable-key8B**, then tuples become 16-bytes and the AVX implementation becomes dysfunctional. In this configuration, algorithms must run with **–scalarsort –scalarmerge** options on the command line.

Our code makes use of the Intel Performance Counter Monitor tool (version 2.35) which was slightly modified to be integrated into our implementation. The original code can be obtained from:

http://software.intel.com/en-us/articles/intel-performance-counter-monitor/

We are providing the copy that we used for our experimental study under **`lib/intel-pcm-2.35`** directory which comes with its own Makefile. Its build process is actually separate from the autotools build process but with the `–enable-perfcounters` option, make command from the top level directory also builds the shared library **`libperf.so'** that we link to our code. After compiling with –enable-perfcounters, in order to run the executable add `lib/intel-pcm-2.35/lib' to your `LD_LIBRARY_PATH`. In addition, the code must be run with root privileges to acces model specific registers, probably after issuing the following command: `modprobe msr`. Also note that with –enable-perfcounters the code is compiled with g++ since Intel code is written in C++.

We have successfully compiled and run our code on different Linux variants as well as Mac OS X; the experiments in the paper were performed on Debian and Ubuntu Linux systems.

Usage and Invocation

The `sortmergejoin` binary understands the following command line options:

```
$ src/sortmergejoins -h
[INFO ] Initializing logical <-> physical thread/CPU mappings...
[INFO ] Getting the NUMA configuration automatically with libNUMA.
Usage: src/sortmergejoins [options]
Join algorithm selection, algorithms : m-way m-pass
       -a --algo=<name>    Run the join algorithm named <name> [m-pas
s]

    Other join configuration options, with default values in [] :
       -n --nthreads=<N>   Number of threads to use <N> [2]
       -r --r-size=<R>     Number of tuples in build relation R <R> [
128000000]
       -s --s-size=<S>     Number of tuples in probe relation S <S> [
128000000]
       -x --r-seed=<x>     Seed value for generating relation R <x> [
12345]
       -y --s-seed=<y>     Seed value for generating relation S <y> [
54321]
       -z --skew=<z>       Zipf skew parameter for probe relation S <
z> [0.0]
       --non-unique        Use non-unique (duplicated) keys in input
relations
       --full-range        Spread keys in relns. in full 32-bit integ
er range
       --no-numa-localize  Do not NUMA-localize relations to threads
[yes]
       --scalarsort        Use scalar sorting; sort() -> g++ or qsort
() -> gcc
       --scalarmerge       Use scalar merging algorithm instead of AV
X-merge
       -f --partfanout=<F> Partitioning fan-out (phase-1, 2^#rdxbits)
 <F> [128]
       -S --numastrategy   NUMA data shuffle strategy: NEXT,RANDOM,RI
NG [NEXT]
       -m --mwaybufsize    Multi-way merge buffer size in bytes,ideal
ly L3-size

    Performance profiling options, when compiled with --enable-perfco
unters.
       -p --perfconf=<P>   Intel PCM config file with upto 4 counters
 [none]
       -o --perfout=<O>    Output file to print performance counters
[stdout]

    Basic user options
        -h --help          Show this message
        --verbose          Be more verbose -- show misc extra info
        --version          Show version
```

The above command line options can be used to instantiate a certain configuration to run the given join algorithm and print out the resulting statistics. Following the same methodology of the related work, our joins never materialize their results as this would be a common cost for all joins. We only count the number of matching tuples and report this. In order to materialize results, one needs to configure the

code with –enable-materialize option. Moreover, to write join input and output tables to file, one needs to configure the code with –enable-persist.

Configuration Parameters

Logical to Pyhsical CPU Mapping

If running on a machine with multiple CPU sockets and/or SMT feature enabled, then it is necessary to identify correct mappings of CPUs on which threads will execute. For instance our experimental machine, Intel E5-4640 has 4 sockets and each socket has 8 cores and 16 threads with the following NUMA topology:

```
$ numactl --hardware
available: 4 nodes (0-3)
node 0 cpus: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60
node 0 size: 130994 MB
node 0 free: 127664 MB
node 1 cpus: 1 5 9 13 17 21 25 29 33 37 41 45 49 53 57 61
node 1 size: 131072 MB
node 1 free: 128517 MB
node 2 cpus: 2 6 10 14 18 22 26 30 34 38 42 46 50 54 58 62
node 2 size: 131072 MB
node 2 free: 128685 MB
node 3 cpus: 3 7 11 15 19 23 27 31 35 39 43 47 51 55 59 63
node 3 size: 131072 MB
node 3 free: 128769 MB
node distances:
node   0   1   2   3
  0:  10  20  20  20
  1:  20  10  20  20
  2:  20  20  10  20
  3:  20  20  20  10
```

Based on the NUMA topology, on our experimental machine logical threads (0..63) are round-robin distributed to NUMA regions. Therefore, on this machine even not using a cpu-mapping.txt file results in a good configuration. On a different machine it might be necessary to create a cpu-mapping.txt file which tells how logical threads from 0 to N are assigned to underlying physical threads. An example for a hypothetical 2-socket 16-thread machine would be the following (#threads mapping-list #sockets):

```
$ cat cpu-mapping.txt
16 0 1 2 3 8 9 10 11 4 5 6 7 12 13 14 15 2
```

This file is must be created in the execution directory and used by default if exists in the directory. It basically says that we will use 16 CPUs listed and threads spawned 1 to 16 will map to the given list in order. For instance thread 5 will run on CPU 8. This file must be changed according to the system at hand. If it is absent, threads will be assigned in a round-robin manner (which works well on our experimental machine).

Warning

NUMA configuration is also automatically obtained if libNUMA is available in the system.

> Otherwise, the system assumes a single-socket machine without NUMA. This can be a problem if not correctly identified. NUMA configuration is used in determining how to allocate L3 cache and sharing of it among threads. Therefore might pose a significant performance drop especially in the **m-way** algorithm if incorrectly done.

Performance Monitoring

For performance monitoring a config file can be provided on the command line with –perfconf which specifies which hardware counters to monitor. For detailed list of hardware counters consult to "Intel 64 and IA-32 Architectures Software Developer's Manual" Appendix A. For an example configuration file used in the experiments, see **`pcm.cfg'** file. Lastly, an output file name with –perfout on commandline can be specified to print out profiling results, otherwise it defaults to stdout.

System and Implementation Parameters

The join implementations need to know about the system at hand to a certain degree. For instance **CACHE_LINE_SIZE** is required in various places of the implementations. Most of the parameters are defined in **params.h** . Moreover, some of these parameters can be passed during the compilation as well as during the runtime from the command line.

Some of the important parameters are: **CACHE_LINE_SIZE**, **L2_CACHE_SIZE** and **L3_CACHE_SIZE** (used for multi-way merge size and can be given from command line).

Generating Data Sets of Our Experiments

Here we briefly describe how to generate data sets used in our experiments with the command line parameters above.

Workload A variants

This data set is adapted from Albutiu et al., where large joins with billions of tuples are targeted. To generate a one instance of this data set with equal table sizes do the following:

```
    $ ./configure
    $ make
    $ ./sortmergejoins [other options] --r-size=1600000000 --s-size=
1600000000
```

Workload B

In this data set, the inner relation R and outer relation S have 128*10^6 tuples each. The tuples are 8 bytes long, consisting of 4-byte (or 32-bit) integers and a 4-byte payload. As for the data distribution, if not explicitly specified, we use relations with randomly shuffled unique keys ranging from 1 to 128*10^6. To generate this data set, append the following parameters to the executable `sortmergejoins`:

```
     $ ./sortmergejoins [other options] --r-size=128000000 --s-size=
128000000
```

Note

Adjust other parameters such as (partitioning fan-out: -f –partfanout=<F> and multi-way merge buffer size: -m –mwaybufsize) according to your machine specs.

Introducing Skew in Data Sets

Warning

The skew handling mechanisms like heavy-hitter detection and such are not fully integrated into this version of the code package. Full support for the skew handling as described in the paper will be integrated/merged from a different branch of the code in the upcoming releases. Please contact us if you have questions on this.

Skew can be introduced to the relation S as in our experiments by appending the following parameter to the command line, which is basically a Zipf distribution skewness parameter:

```
$ ./sortmergejoins [other options] --skew=1.05
```

Testing

Our code uses **Check** unit testing framework for C code (version 0.9.12, provided in lib/). Latest version of Check can be obtained from:

http://check.sourceforge.net/

Various parts of the implementation such as functionality of sorting, merging, multi-way merging and partitioning are automatically tested by unit tests contained in **tests/** folder. After installing **Check** in your system, the procedure is as simple as issuing the following command from the top directory:

```
$ make check
```

Other than that, there are some scripts provided to test features like join execution and materialization results which can also be found in **tests/** folder.

Author

Cagri Balkesen cagri.balkesen@inf.ethz.ch